Efficient Tree Construction for Multiscale Image Representation and Processing

Jiří Havel, François Merciol, Sébastien Lefèvre

Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Jiří Havel · François Merciol · Sébastien Lefèvre

Efficient Tree Construction for Multiscale Image Representation and Processing

Received: date / Revised: date

Abstract With the continuous growth of sensor performances, image analysis and processing algorithms have to cope with larger and larger data volumes. Besides, the informative components of an image might not be the pixels themselves, but rather the objects they belong to. This has led to a wide range of successful multiscale techniques in image analysis and computer vision. Hierarchical representations are thus of first importance, and require efficient algorithms to be computed in order to address real-life applications. Among these hierarchical models, we focus on morphological trees (e.g., min/max-tree, tree of shape, binary partition tree, α-tree) that come with interesting properties and already led to appropriate techniques for image processing and analysis, with a growing interest from the image processing community. More precisely, we build upon two recent algorithms for efficient α-tree computation and introduce several improvements to achieve higher performance. We also discuss the impact of the data structure underlying the tree representation, and provide for the sake of illustration several applications where efficient multiscale image representation leads to fast but accurate techniques, e.g. in remote sensing image analysis or video segmentation.

1 Introduction

Mathematical morphology has long been a provider of interesting hierarchical image representations, mainly by trees, e.g., component tree [22], min and max-tree [35], binary partition tree [34], etc. The max-tree (and its respective counterpart, min-tree) has been widely used due to its nice properties as well as the availability of efficient algorithms to first compute the tree from an image, and second process the tree (e.g., with a filtering to remove irrelevant nodes), thus leading to the processing of the underlying image.

The min and max-tree can be combined together to form a tree of shapes [42]. In the tree of shapes, child nodes represent parts of a larger object. Another improvement of min/max-trees uses second generation connectivity [28], where the contents of the tree and its shape are defined by two independent images.

Recently, a new image model, namely the α-tree [29], has been introduced to avoid relying on an ordering relation among image pixels. This model is a hierarchical representation of the quasi-flat zones of an image, and as such, relies on local dissimilarities α. This model already led to successful attempts in exploration of remote sensing data [27] and image/video segmentation [19]. Due to the locality of the α-tree, an (α, ω)-tree was introduced [39, 40]. (α, ω)-tree uses nonlocal infomations to prevent forming of excessively large components.

There is large amount of algorithms for construction of min/max-tree [7]. For α-tree construction, two algorithms have been recently introduced. One [12] focuses mainly on utilizing parallelism on modern multicore processor. The other [23] is based on relationship [9] of various trees (binary partition tree by altitude ordering, α-tree, min/max-tree, wathersheds, ...) to the minimum spanning tree. In this paper, we fuse these approaches. Additionally, the relationship of min/max-tree and the minimum spanning tree will be slightly extended to include mask based second generation connectivity.

Section 2 will briefly describe various component trees and their relationship. Section 3 compares some possible representations of the component tree and describes the combined algorithm for construction of α-tree. In section 4, all algorithms are compared in terms of performance and section 5 illustrate the relevance of α-tree with various applications facing large data volume.

J. Havel
Brno University of Technology, Czech Republic
Tel.: +420 54114-1474
Fax: +420 54114-1270
E-mail: ihavel@fit.vutbr.cz

F. Merciol, S. Lefèvre
Université de Bretagne-Sud, IRISA, France
E-mail: {francois.merciol,sebastien.lefevre}@univ-ubs.fr

2 Connected Component Trees

2.1 Definitions

All connected component trees represent an image as a hierarchy of regions i.e., connected sets of pixels. For the purpose of the connected component trees, the image is a undirected graph $G = (V, E, I)$, where the vertices V correspond to the image pixels (so either $V \subset \mathbb{Z}^2$, or an injective mapping pixel : $V \to \mathbb{Z}^2$ can be constructed) and the edges E connect pixels as described by the incidence relation $I \subseteq V \times E$. For the purpose of this paper, the sets V and E are some identifiers of vertices and edges, so the sets are disjoint. The exact nature of these identifiers is not important. We will also disallow hypergraphs (and loops) ($\forall e \in E : |\{v \in V \,|\, vIe\}| = 2$) and multigraphs ($\forall v_1, v_2 \in V : |\{e \in E \,|\, v_1 Ie, v_2 Ie\}| \leq 1$). This definition allows to make subgraphs by subsetting V and E and keeping I intact. The degree of vertices depends on the image connectivity. For square pixels and grid, 4- or 8-connectivity is most common. 6-connectivity is sometimes used, but is typical for hexagonal pixels and grid. The degree of a vertex (with the exception of image border) is equal to the image connectivity.

A *connected component tree* (CCT) is a rooted tree whose each node corresponds to a connected component (CC). A CC is a subgraph whose all vertices are all connected by a path and no additional vertex can be added to it. To get a hierarchical tree and not only a single partition, CCT must contain CCs of various image subgraphs $G_i = (V_i \subseteq V, E_i \subseteq E, I)$, or shortly $G_i \subseteq G$. Additionally, a tree node (i.e., a subgraph of G) must include the union of all its children. Usually, only the tree root is a CC of G (usually the whole image).

The hierarchy of subgraphs can be created by introducing a weight function $w : V + E \to W$, that assigns weights to all vertices and edges of the image graph. The weight functions for vertices and edges can be different and the weight sets too, but we use only one for the sake of simplicity. The set of weights W is usually a subset of $\mathbb{R}$ and must have total ordering. A set of CCs for a weight $w_i \in W$ is a set of CCs of subgraph G_i :

$$V_i = \{v \in V \,|\, w(v) \leq w_i\} \tag{1}$$
$$E_i = \{e \in E \,|\, w(e) \leq w_i\} \tag{2}$$

These CCs form a level of the CCT that corresponds to the weight w_i and contain subsets of vertices and edges of G. For simplicity, we will write that these CCs and the tree level have the weight w_i. A vertex v first appears in the tree level with weight $w(v)$ and is present in all parent levels all the way to the root.

A CCT can provide a hierarchy of image segmentations. A segmentation is a decomposition of an image into disjoint groups of pixels (segments or regions), whose union is the complete image. To provide a segmentation, the tree must contain all image pixels at the level $\min(W)$. If a tree has vertices at a different level, the segmentation can be constructed by setting $w(v) = \min(W)$, i.e. by pulling the pixels as separate leaves. Such segmentation adds single-pixel segments for all vertices that appear on a level closer to the root.

Cousty et. al. [9] have shown, that the edge weights alone are sufficient for construction of various morphological trees such as

- Binary partition tree by altitude ordering.
- Min and max trees (without singleton components).
- α-tree, i.e. a hierarchy of flat zones.
- Hierarchy of watershed cuts.

The addition of vertex levels allows to construct additional trees such as

- Min and max trees including singleton components.
- Mask based second generation connectivity trees.

The min and max-trees are the basic connected component trees. The leaves of min-tree (resp. max-tree) contain image local minima (resp. maxima) and the root the whole image. A min-tree has $w(v)$ equal to the pixel value and

$$w(e) = \max\{w(v) \,|\, vIe, v \in V\} \tag{3}$$

i.e., a subgraph contains all edges that connect its vertices. Its dual construction is called a *max-tree* and can be constructed similarly by negating the weights and constructing a min-tree. The min and max-trees share one important limitation. The pixel values must have *total ordering* so they can be used as weights. While this is trivial for greyscale images or for a raster of responses of some classifier, multivariate data such as color values or spectral signatures have no reasonable comparison function. Even if it is possible to add artificial ordering to the colors, the results are of a limited use [37]. One of the tree representations that do not require total ordering of pixel values is the recently introduced α-tree.

The *α-tree* is a hierarchy of one kind of quasi-flat zones [39] – the α-zones (or α-connected components). It is based on a dissimilarity metric $d : V \times V \to \mathbb{R}$ (or a subset of $\mathbb{R}$, such as common $\{0, \ldots, 255\}$) between two pixels. An α-zone is a set of pixels, where every two pixels are connected by a path. The dissimilarity of every two adjacent vertices on that path must be less or equal to the value of α.

$$\alpha\text{-}\mathcal{Z}(x) = \{x\} \cup \{y \,|\, \exists \pi_\alpha(x \rightsquigarrow y) \\ \forall v_i \in \pi_\alpha(x \rightsquigarrow y) : v_i \neq y \Rightarrow d(v_i, v_{i+1}) \leq \alpha. \tag{4}$$

With $\alpha = 0$, this definition leads to flat zones [36], i.e. the sets of equal (w.r.t dissimilarity metric d) pixels. Ouzounis and Soille in [29] used the α-zones to construct the α-tree based on a partition pyramid of V. Levels of the α-tree correspond to the values of α and the tree components represent the α-zones. When the edge weights are

equal to the dissimilarity metric d between the connected pixels

$$w(e) = d(v_1, v_2), v_1 I e, v_2 I e, v_1 \neq v_2 \quad (5)$$
$$w(v) = \min(W) \quad (6)$$

then the constructed component tree is the α-tree, where the weights of tree levels are the values of α. Let us note that the α-tree can be built also by using a similarity metric instead of a dissimilarity one. Similarly to the max-tree construction, the weights can be negated or the tree can be constructed from subgraphs

$$V_i = \{v \in V \,|\, w(v) \geq w_i\} \quad (7)$$
$$E_i = \{e \in E \,|\, w(e) \geq w_i\}. \quad (8)$$

In this case, the weight of tree root is $\min(W)$. The rest of this paper will deal only with the dissimilarity metric and trees with root level $\max(W)$.

Soille [39], and later Soille and Grazzini [40], have also introduced new α-zone definitions with additional constraints through the constrained connectivity paradigm. Among them, the (α, ω)-zones rely on both a local range to be compared to α and a global range to be compared with ω:

$$(\alpha, \omega)\text{-}\mathcal{Z}(x) = \bigvee \{\alpha_i\text{-}\mathcal{Z} \mid \alpha_i \leq \alpha \ \vee \ \mathrm{R}(\alpha_i\text{-}\mathcal{Z}) \leq \omega\} \quad (9)$$

with $\mathrm{R}(\cdot)$ denoting the global range of the quasi-flat zone. The global range constraint works against the purely local behavior of the simple α-zones to alleviate the chaining effect faced with all single-linkage procedures. Such quasi-flat zones have been already used in image segmentation through an interactive framework [49]. However, this requires to set predefined values for α and ω parameters, which is not an easy task in practice. The (α, ω)-zones are the largest α-zones that satisfy the condition $\mathrm{R}(\alpha_i\text{-}\mathcal{Z}) \leq \omega$. To modify the α-tree to use (α, ω)-connectivity, it is necessary to calculate the range $\mathrm{R}(\cdot)$ for each node of the tree. If the value of $\mathrm{R}(\cdot)$ is increasing (i.e. $Z_1 \subseteq Z_2 \rightarrow \mathrm{R}(Z_1) \leq \mathrm{R}(Z_2)$), the value of $\mathrm{R}(\cdot)$ can be used to assign different levels to the trees. This relevelling of the tree can not split tree nodes, only change the levels and possibly merge certain nodes, if the value $\mathrm{R}(\cdot)$ is the same both for the parent and child node. Therefore it can be done by postprocessing the simple α-tree, following the tree transformation proposed in [3].

With vertex weights in addition to edge weights, it is possible to describe second generation connectivity too. In second generation connectivity, connected components are localized in a graph modified by some operator.

This operator can either extend or shrink the connected components. The mask-based second generation connectivity [28] merges both kinds of the operators. Mask-based connectivity uses two images. An input image X and a mask image M that can be either derived from X by some operator (e.g., dilation or erosion) or acquired by a completely different way.

When X and M are binary images (or threshold decompositions) i.e. sets of points, the connected component for a point x is

$$\mathrm{CC}_x^M(X) = \begin{cases} \mathrm{CC}_x(M) \cap X & \text{if } x \in X \cap M \\ \{x\} & \text{if } x \in X \setminus M \\ \emptyset & \text{otherwise} \end{cases} \quad (10)$$

. Informally, if the point x is both in original and mask image, connected component is constructed in the mask image and then vertices that are not in original image are removed. If the point lies in original image only, its connected component is a singleton.

The images X and M will be described by two weight functions w_X and w_M for a common image graph $G = (V, E, I)$. All connected components except singletons can be constructed using edge weights only. Since the existence of singletons is dependent on X, only edge weights need to be taken from M. Also since the connected components are constructed in M, only vertex weights from X will be used. This way, the original and mask image can be merged to a single weighted graph.

For example,

$$w(v) = w_X(v) \quad (11)$$
$$w(e) = \max\{w_M(v) \,|\, vIe, v \in V\}, \quad (12)$$

specify a second generation min-tree based on vertex weights w_X and w_M. The mask image specifies the shape of a min-tree minus the singleton components. The image X specifies the levels, where every vertex lies in the final tree. If $w_X(v) < w_M(v)$, the vertex is moved down into a singleton component. If $w_X(v) > w_M(v)$, the vertex is moved up in the tree and leaves a hole in the original component.

2.2 Applications

Hierarchical representations reviewed so far have been used to solve various problems in image analysis and processing.

Image simplification can be achieved through tree pruning. When irrelevant nodes are deleted from the tree, the corresponding details are removed from the associated image. This task has been successfully achieved on greyscale images with the tree of shape [21], and later on natural color photographs and satellite multispectral images with the α-tree [39].

Tree pruning can also lead to image segmentation. Indeed, and as it has been already noticed, tree leaves provide an image partition. While the leaves in an original tree can represent singletons or isolated pixels, pruning the tree will result in the removal of irrelevant branches, and the definition of new tree leaves. Such leaves then describe not singletons anymore but rather larger regions,

which still compose a partition of the input image. Performing image segmentation through analysis and pruning of the underlying tree requires to define some pruning criteria. In a fully unsupervised framework (i.e., without any user intervention), this is far from being trivial and the question of selecting an optimal cut of the tree is still open [38]. Nevertheless, automatic segmentation of color images or even video sequences has already been achieved [19]. Another strategy to exploit trees for image segmentation is to involve the user in the loop. By selecting relevant nodes, the user can then drive the segmentation towards the expected result, leading to interactive methods for video sequences [19], medical images [31], or historical document images [30].

Instead of pruning the tree, one can also consider selecting relevant information from the tree i.e., specific nodes or sets of nodes. This helps to achieve object detection, as successfully demonstrated with the Binary Partition Tree [46]. Computing some features from these significant nodes (w.r.t. a given criterion) also allows for feature extraction, and trees can then be used for extraction of stable regions (MSER) [24]. This opens the way for addressing content-based image retrieval with such hierarchical representations, as it has been proposed with scenery images in [41].

As an efficient and compact model of an underlying image, trees have been considered to analyse large remotely-sensed images. Even with a tree built from a single greyscale image, it is possible to perform damage assessment (rubbish detection) [29] or to provide an interactive framework for assisted selection of some areas of interest, such as refugee tents in the context of human crisis management [26]. Remote sensing has also been addressed for supervised or unsupervised image classification, which can be achieved through the definition of appropriate criteria for modeling the tree nodes (vertices and edges), and a subsequent tree pruning strategy [45]. Let us note that the tree model can help to make the process more independent from the nature of input data (e.g., both synthetic aperture radar (SAR) and hyperspectral images are considered in [1]). Furthermore, tree construction can rely on prior knowledge to improve the tree model, e.g. using labeled samples in a semi-supervised framework for representing hyperspectral images [18, 15].

Since a tree is a compact description of an image, it can also help to achieve image comparison or matching with a reduced cost. This is especially useful for motion detection and object tracking [21], or image registration [20].

In this paper, we will illustrate the interest of tree modeling of images and videos with two applications where data to be handled are rather large: remotely-sensed image analysis and interactive video segmentation (Sec. 5). But as it has been shown with the various applications reviewed in this section, the efficient algorithms proposed in this paper can be used in a much wider range of scenarios.

3 Tree construction

This section will present an algorithm for the construction of trees from Sec. 2. This algorithm is based on the parallel α-tree construction algorithm from Havel et. al. [12] and combines it with another recent work from Najman et. al. [23].

Compared to Najman's algorithm [23], our previous algorithm [12] constructs α-tree directly, instead of post-processing a BPT. Since α-trees can have significantly less nodes compared to the BPT, it is beneficial to skip creation of unnecessary nodes. The number of α-tree nodes compared to number of BPT nodes is extremely variable, but 40-70% seems to be good although very rough estimate (see Fig. 1 in Sec. 4).

The separate MST construction in [23] provides better decoupling of the tree and the root-finding structures than ours [12]. The proposed algorithm takes benefit of the advantages of these two approaches by using this decoupling while creating less redundant nodes.

Additionally from merging [23] and [12], it also allows construction of second order trees by interleaved insertion of vertices and edges to the constructed tree.

In the sequel, every part of the construction will be first described for an edge weighted graph and then the algorithm will be extended for vertex weights.

3.1 Data structures

To represent the morphological tree, we need to store an oriented forest. A morphological tree consists of leaves (image pixels) and inner nodes (connected components). Empty nodes that represent empty components can exist. These are in fact leaves, but will be treated as empty inner nodes. A node that has at most one *nonleaf* child will be called *degenerate*.

Several representations of the tree are possible, and we consider here the following options:

- Array of parent indices as used in [23] (*arrays* in short).
- Dynamically allocated structures linked by pointers as used in [12] (*structs* in short).

The *arrays* represent a tree node by an integer. The main part of this representation is an array *parent* of parent nodes. parent(i) contains the index of the parent of node i. A root node can be marked by several ways such as parent(i) $= i$ or parent(i) $= r$, where r is other to any valid node number. In the array tree, node numbers in $[0, |V|[$ represent the tree leaves and the nodes in $[|V|, N[$ represent the components. When $i \leq$ parent(i), this representation allows simple top-down or bottom-up

tree traversal. When every nonleaf node has at least two children, then $N = 2|V| - 1$ and the constant r can be set to N.

The *structs* represent a tree by a pointer to a dynamically allocated Node structure, that contains a pointer to a parent node and the necessary pointers for maintaining linked lists of child nodes. Nonleaf node also contains linked list of its children. The number of leaf nodes is usually known, so the leaves can be allocated as one large memory block.

For tree construction, the forest representation must allow several operations:

- findRoot : Vertex → Node for finding a root from a vertex;
- leaf : Vertex → Node for finding leaf node corresponding to a vertex;
- newRoot : () → Node for adding a new root node to a tree;
- attach : Node × Node → () for attaching a node to another one as a child;
- merge : Node × Node → Node for merging of two root nodes into one;
- reattach : Node × Node → () that detaches node from its previous parent and attaches to new one.

For the sake of simplicity, the whole forest on which the functions operate will not be passed as a parameter.

The function *findRoot* is critical to the tree construction performance. The tree representations are not suitable for fast root lookup, so an auxiliary structure must be used. Tarjan Union-Find [43] (Q_{ct} in [23]) is a commonly used building block for finding roots in a forest and subsequent merging of its trees. This function could be also naively implemented by traversal up from a node returned by *leaf*.

In code by Najman et al., union-find uses two arrays. One for storing node parent and one for tree heights (for union by rank). The parent array stores integers lower than $|V|$ for nonroot nodes and the rank array stores the tree height(¡ $\log_2 |V|$) for root nodes. It is possible to combine these arrays together (limiting $|V|$ to $2^N - N$ instead of 2^N is usually ok), so values $< |V|$ mark child node and $\geq |V|$ mark root.

The function *newRoot* adds a new root node to the forest. In *arrays* it simply increments the node counter and returns its previous value. In *structs* it allocates and initializes new node.

The function *merge* merges two root nodes. Merging of two nodes requires to iterate over children of one node and reconnect them. Since the nodes can be merged repeatedly, node merging could lead to $O(N \log N)$ reconnections if the node merges form a balanced tree. When the function *merge* only marks the nodes for removal and the nodes are removed in a postprocessing step, the number of reconnections can be lowered to $O(N)$. The postprocessing steps significantly differ in case of *structs* and *arrays* and will be described later.

Let us emphasize that selection of proper tree representation depends on the problem.

The *arrays*

- use significantly less memory; the *structs* allocate many small objects so the alignment and allocation overhead is significant; also on 64b architectures, the pointer size adds to memory requirements of *structs*;
- lead to a simpler and less verbose code;
- allows for very easy basic tree traversal, but only layer by layer of the tree. For different traversals an array of child indices must exist for each component;
- allow node removal that leaves holes in the array;

The *structs*

- have more flexible structure; nodes can be simply added or removed at various places in the tree; In *arrays*, the insertion can lead to a shift of the whole array;
- do not need any upper bound on the number of tree nodes; *arrays* may need reallocation if such number is exceeded during construction.

3.2 Basic construction

The tree is constructed from a list of weighted edges and optionally also a list of weighted vertices. These two lists are first ordered by increasing weights. The weights of the image graph vertices and edges can have various datatypes. To make the construction as generic as possible, the weights will not be used directly. Several predicates will be used instead. These predicates can internally compute the weights and levels and compare them, but from the construction viewpoint, only the comparison does matter.

The tree is constructed from sorted lists. The sorting method is not important here. However when edges are sorted by any $O(N \log N)$ method, then whole complexity can no longer be pseudolinear. However for all reasonable weight types, $O(N)$ algorithms exist (e.g., bucket sort, if the weights are small integers or more generic radix sort). These faster methods are usually also stable compared to methods like quicksort. Stable sorting together with regular image scanning has a large impact on tree construction (evaluated in Sec. 4.1).

Each edge of the input list is processed by Alg. 1. The algorithm uses $<_{ne}$ to determine if nodes lie in the same layer, where the edge belongs. The merge needs to handle four cases:

- new root is created (both r_1 and $r_2 <_{ne} e$);
- r_1 is attached as a child to r_2, or vice versa;
- the roots are merged to one.

This minimizes the amount of nodes that are created and then removed by merge function. The comparison by $<_n$ simplifies the structure since it ensures

$$r_1 <_{ne} e \implies r_2 <_{ne} e. \tag{13}$$

Algorithm 1 Edge insertion

Require: Edge e, connecting vertices v_1 and v_2
Ensure: Common root r, modified forest
  $r_1 := \text{findRoot}(v_1)$
  $r_2 := \text{findRoot}(v_2)$
  if $r_1 \neq r_2$ **then**
    if $r_2 <_n r_1$ **then**
      swap(r_1, r_2)
    end if
    if $r_1 <_{ne} e$ **then**
      $r := \text{newRoot}()$
      attach(r_1, r)
    else
      $r := r_1$
    end if
    if $r_2 <_{ne} e$ **then**
      attach(r_2, r)
    else
      $r := \text{merge}(r_2, r)$
    end if
  end if

so r_1 will never be lifted to the level of e when r_2 is already there. $<_{ne}$ and $<_n$ must together ensure that

$$n_1 <_n n_2 \implies (n_2 <_{ne} e \implies n_1 <_{ne} e). \tag{14}$$

If these predicates internally compare the edge weight and the weights stored in tree nodes, then this condition is obviously true. The predicate $<_{ne}$ can affect the shape of the constructed tree. If the predicate is constantly true, then the algorithm creates a binary partition tree by altitude ordering. If the predicate is constantly false, then the tree construction does simple component labeling. Also, $l <_{ne} e$ must return true for all leaves and edges.

The predicate $<_n$ can generate many more equivalence classes than the number of tree levels. For example, in *arrays*, this predicate can directly compare the node indices, since nodes closer to tree leaves have lower indices in the array. Also $<_{ne}$ can be simplified in *arrays*. When the algorithm processes last edge of a tree layer, it can store index of the last node and $<_{ne}$ can also compare node indices instead of edge weights.

Algorithm 2 Vertex insertion

Require: vertex v
Ensure: Root r, modified forest
  $r := \text{findRoot}(v)$
  if $r <_{nv} v$ **then**
    $c := r$
    $r := \text{newRoot}()$
    attach(c, r)
  end if
  reattach(leaf$(v), r)$

When the tree is built also from a list of vertices, the tree leaves are linked directly to their proper place in the tree during the vertex insertion. This is described by Alg. 2. This algorithm uses another predicate $<_{nv}$ – comparison of tree node and a vertex. For each vertex a root is found and optionally lifted to the vertex level and then the tree leaf is reconnected to it. Similarly to the predicate $<_{ne}$, $l <_{nv} v$ must return true for all leaves and vertices.

The reattaching of a node can lead to a tree that contains degenerate and empty nodes when all vertices are lifted from previously nonempty node. This can be detected immediately or the tree can be postprocessed and such nodes removed after the construction.

Algorithm 3 Tree construction

Require: Ordered lists of edges E and vertices V
  initForest()
  while $E \neq \emptyset \wedge V \neq \emptyset$ **do**
    if $V \neq \emptyset \wedge (E = \emptyset \vee \text{head}(V) <_{ve} \text{head}(E))$ **then**
      insertVertex(head(V))
      $V := \text{tail}(V)$
    else
      insertEdge(head(E))
      $E := \text{tail}(E)$
    end if
  end while

Algorithm 3 shows, how are the edge and vertex insertion put together. The algorithm interleaves the insertion of elements from the sorted vertex and edge list. The structure of these lists is hidden behind classical functions *head* that takes first element of the list and *tail* that returns the list without head.

When initialized by function initForest, the forest consist of $|V|$ separate leaves (so $|V|$ trees) for image pixels. Edge insertion can lower the number of trees by 1 if it joins 2 disjoint trees. Vertex insertion only reconnects leaf node to a higher level of the tree, so it doesn't change the tree count.

When the insertion of edges and vertices is properly interleaved, the algorithm can reuse the same root finding structures for both. This interleaving is driven by a comparison predicate $<_{ve}$. The tree must be built layer by layer and each layer is first built from edges and then the vertices are inserted. It is possible to first lift vertices to a new layer, but this could lead to creation and merging of unnecessary singletons.

3.3 Delayed merges

As mentioned previously, the function *merge* does not directly reconnect the child nodes of the one absorbed node, but only marks a node for removal. The exact implementation for *structs* and *arrays* is different, so it will be described separately.

For *structs*, each node structure contains pointers for inclusion in a linked list. Therefore, a node labeled for removal is removed from the list of children of its parent node and inserted into a separate list. At the end of the

Algorithm 4 Merging in *structs*

```
Require: List n_0, ... n_N of nodes to be removed
  for n := n_N downto n_0 do
    if n.parent has no children then
      n.parent := n.parent.parent
    end if
    relink children of n to n.parent
  end for
  delete nodes n_i
```

processing, each element of the list has its parent either in the tree or further in the list. If the list is processed from the lastly added node to the first one, then children are visited after parents. This is shown by Alg. 4. If the parent pointer points to a node marked for removal, that node was already visited, is empty and its parent pointer points to a node in the tree. This test can be also done by comparing the stored weights. When vertices are inserted into tree, nodes marked for removal can be emptied between edge and vertex insertion for each tree level. By interleaving these merges, removed nodes can be immediately reused in new layers. This is especially useful for parallel code since memory allocation can lead to serialization and this node reuse is effectively free.

Algorithm 5 Merging in *arrays*

```
Require: Leaf count L, node count N, arrays parent and
  merges
  for i := N − 1 downto L + 1 do
    merges[i] := merges[merges[i]]
  end for
  for i := 0 to L − 1 do
    parent[i] := merges[parent[i]]
  end for
  for i := L to N − 1 do
    parent[i] := merges[parent[i]]
    if i ≠ merges[i] then
      Mark node i as empty.
    end if
  end for
```

For *arrays*, nodes do not have lists of children ready. An integer array *merges* will be used instead. For L vertices, both *parents* and *merges* arrays will need $2L - 1$ elements (for full binary tree). The root node i is marked by $\text{parent}[i] = i$. Then

- $\text{merges}[i] = i$ means, node i should be kept.
- $\text{merges}[i] = j$ means, node i should be merged into node j (which may be also marked for removal).

The merging process is then described by Alg. 5. First loop finds the correct node for merging. The next two loops reconnect all leaves and non-leaves to right parents. When $\text{merges}[i] = i$, the assignments do not change values. The third loop also marks removed non-leaf nodes.

3.4 Tree merging

The previous algorithm constructs complete tree at once. It is possible to build partial trees for parts of the input and then merge them together. This can be used for several purposes. First, partial trees can be built in parallel so the algorithm can utilize multicore processors. Second, by adding new partial trees and removing older ones, it is possible to incrementally update the tree for a graph that does not need to be held in memory at once. For example, a videosequence can be processed by such "sliding" tree.

This parallelization strategy is similar to parallel max-tree construction by Matas et al. [17] and Wilkinson et al. [50]. The difference lies within the process that merges two paths when an edge connects two partial trees. Partial trees are constructed by the single-threaded algorithm. Edges that connect the subgraphs must be inserted by an algorithm with greater asymptotic complexity, so their number must be as low as possible.

The edges that connect the partial image areas are processed sequentially. If the partial construction creates only one tree for each image part, the first edge processed during merge joins these two trees and all other edges merge only inner parts of the tree. This observation can be used to simplify the tree merge.

Algorithm 6 Inner tree merge.

```
Require: Tree T and edge e
Ensure: Modified tree T
  (c_1, p_1) = findTransition(e.p_1, e)
  (c_2, p_2) = findTransition(e.p_2, e)
  n = node(e, c_1, c_2)
  while p_1 ≠ p_2 do
    if p_2 <_n p_1 then
      swap(p_1, p_2)
    end if
    if p_1 <_n p_2 then
      p'_1 = p_1.parent
      p_1.attach(n)
      n = p_1, p_1 = p'_1
    else
      p'_1 = p_1.parent, p'_2 = p_2.parent
      n' = merge(p_1, p_2)
      n'.attach(n)
      n = n', p_1 = p'_1, p_2 = p'_2
    end if
  end while
```

Algorithm 6 merges inner parts of a tree together. The function $(c, p) = \text{findTransition}(l, e)$ traverses the tree from the leaf l to the root and finds a node pair (c, p) such that c is direct child of p, $c \leq e$ and $\neg(p \leq e)$. New parent node for children is inserted as before by function node. The following while cycle traverses the tree up to the point where the paths converge and merges them similarly to a zipper. If both paths have a node with same level, these nodes are merged together. This tree merging is identical to the tree merging done in [12].

This simple merging has one issue. The functions findTransition always traverse complete tree, even when the paths join below the level of the inserted edge. The traversal can be improved by a synchronous lookup where both paths from leaf to root are traversed simultaneously and when the paths join, the lookup ends. The pseudocode is not included. The traversal repeatedly does one step up in the tree and switches the paths if the active one is higher in the tree.

Another important implementation issue lies in the merge function. When two nodes on the same level are merged together, children of the smaller one must be inserted into the larger. Otherwise the whole merging process has worst case quadratic complexity. The effect of this can be so intense, that it can completely negate all speedup gained by parallel processing. This effect will be measured and discussed in Sec. 4.

3.5 Practical concerns beyond 8-bit image coding

The various algorithms introduced so far are generic. While they might be applied on any kind of data, they are especially adapted to common image coding standards, i.e. 8-bit coding for greyscale image which leads to 256 levels in the tree (since both vertices and edges are defined in $[0, 255]$). In the case of 24-bit color images, the same tree height can be assumed under the following conditions: a) either the vertices are coded on 8 bits, which is made possible by a color quantization, b) either the edges are coded on 8 bits, which is the case when using an L_∞ norm as a dissimilarity metric between pixel values (as in [19]). Even if the algorithm needs to go through each of the 256 levels during the edge insertion process, that is still a neglected constant w.r.t. the number of pixels in the image.

Let us note however that this is not anymore the case when dealing with a much higher number of levels, e.g. with HDR imaging or if the dissimilarity metric takes values in a real space (e.g., simply using an Euclidean distance). Increasing tree height directly results in a higher merging cost. Besides, other issues related to tree analysis might be raised and will be discussed here.

3.5.1 High-dynamic-range imaging

High-dynamic-range (HDR) imagery aims to capture a greater dynamic range between the lightest and darkest areas of an image than with usual 8-bit (grayscale) or 24-bit (colors) coding. Standard HDR implementations consider 16-bit (half precision) or 32-bit floating point values to describe each image pixel. The overall data size for a given pixel can then be equal to 96 bits, or only 32 bits (RGBE or LogLuv). Other very specific imaging devices (e.g., in medical, biological, satellite, or astronomical imaging) may come with a depth higher than 8 bits (see Sec. 5.1) and as such, may be also considered as HDR images. All these images lead to a very large range of values for the vertices of their underlying tree model.

As already stated, the intrinsic complexity of the tree management process directly depends on the number of levels within the tree. While this number is usually 256 levels with standard greyscale images, from a theoretical viewpoint it can be as high as 4.3 billions (32 bits) or even $8 \cdot 10^{28}$ (96 bits). This is definitely unachievable with the current computing facilities. Fortunately, the tree construction comes with higher bounds (image size for tree height and twice for the number of nodes) i.e., much lower than the theoretical bounds. We will specifically evaluate the computational cost w.r.t. tree height in the next section.

In the specific case of HDR images, the advantage of modeling the image through a tree is twofold. On the one side, it provides an underlying structure that can be easily manipulated to select (either manually or automatically) the areas in the image where the contrast as to be highlighted, assuming the display screen is limited to an 8-bit range. Indeed, efficient access to image regions through the tree structure enables to prune the tree and to select areas (i.e., subgraphs) of interest at a low cost w.r.t. pixelwise analysis of the whole image as in state-of-the-art algorithms. These regions are given a larger contrast range to the detriment of other regions. The overall process can be interactive and enables real-time visualization of HDR images. On the other side, and if particular attention has to be paid to the memory footprint, it allows an iterative construction process of the tree model. To explain, let us consider that the tree is built with only the 8 most significant bits (MSB) of an HDR image (i.e., 256 levels). The user can visualize the image, browse within the tree structure and select the leaves of interest, for which a greater contrast is desired. For each leaf, it is possible to build a new tree rooted from the leaf itself, and of depth equal to the number of levels (256 if the 8 following MSB are considered). The image contrast is updated, increased in the area described by the leaf to represent all levels from the new tree, and lowered for the other parts of the image. Of course there is no limitation and the process can iterate over the whole value range, providing an interactive way for visualizing HDR image content.

Let us note however that such a strategy may lead to some modification in the tree topology. In the case of of α-trees or other trees built from edge analysis, considering only the MSB instead of the full data range changes the computation of edge values, and thus the way nodes are associated in the bottom-up tree creation process. While this issue is not faced with the trees built from vertex values, exploring such trees might be cumbersome since the strategy is well-adapted to leaf expanding only. If inner nodes have to be further explored, it requires to

keep considering the MSB while looking simultaneously to the finer contrast range.

3.5.2 Real valued edges or vertices

When considering real or floating-point coding of pixel or dissimilarity values (i.e., resp. vertex and edge values), one has to face the possible infinity of levels within the tree (the actual limit is given by the computer coding precision). Despite this possible infinite number of values for edges and vertices, we know that the number of nodes in the tree is finite and bounded by twice the image size. In the degenerate case, the tree height is bounded by the image size. However, since edges or tree levels are represented by real values, it is not possible to provide a direct access to a given level of the tree. An iterative scan is required, either from the root or the leaves. Navigation within a very deep tree may require the same computational cost as a standard pixelwise scanning, thus preventing interactive image analysis and processing.

To alleviate from this issue, we suggest the following strategy. While the tree levels are theoretically infinite, their number is practically limited by the image size. These levels are taken from a subset of the set of edges computed from the image. It is thus possible to allocate an array to contain all the image edges (with size equal twice the image size for 4-connectivity), scan the image pixels, and store the edge values within the array. When building the tree, it is possible to update the array by adding links to the different tree levels, thus allowing an efficient access to the tree levels with a single indirection. A direct access can even be made possible if hashing techniques are involved. Nevertheless, this technique comes with an important memory burden (and a significant computational effort when scanning the image) due to the additional array, even if the initial size (e.g., twice the image size) can be reduced after tree construction by keeping only relevant cells or levels (i.e., at most the image size).

4 Experiments

In order to assess the performance of the algorithms proposed in Sec. 3, we settle a set of experiments using some standard image databases:

- Berkeley segmentation dataset (BSD) [16];
- INRIA Holidays dataset (IHD) [13];
- A true orthophoto acquired from aerial imagery (TOP) [10].

The BSD dataset is designed to evaluate image segmentation methods [16], but the actual segmentation is out of scope of this paper (while it is partly addressed on massive data in Sec. 5). All images in this dataset have the same number of vertices and edges, so the differences in the constructed trees can depend only on the image content. Since the images are quite small (481×321 pixels), this dataset will be used mainly for evaluation of the structure of various kinds of trees. For performance measurements, we rely on two other datasets. The IHD dataset provides wide range of high resolution images (1024×768 pixels) of natural and man-made scenes and has been designed for image retrieval tasks [13], while this application is not addressed in this paper. The TOP image is a very large orthorectified photograph (20250×21300 pixels) built as a mosaic of color aerial images. It is extracted from the Vaihingen dataset from the ISPRS benchmark on urban object detection and 3D building reconstruction [10].

We address performance evaluation through various experiments. The effects of order of edge extraction and the stability of the sorting method on the subsequent tree construction are first evaluated. The performance of single-threaded tree construction is then compared to previously published algorithms. The two ways to implement the component tree are also compared in terms of performance. Finally an experimental evaluation of the parallel tree construction is provided.

We measured the effect of union-find modification (i.e., packing root rank into the parent array) on the IHD data. The construction of a spanning tree was accelerated by 9% on average. For the whole tree construction, this is a minor speed gain, but practically free (e.g., changing the maximum vertex count from 2^{32} to $2^{32}-32$).

4.1 Order of edge insertion

The order in which the edges are inserted into the tree affects the structure of the binary tree, but not the α-tree, since nodes with 3 and more children can be split to binary nodes in multiple ways. However the edge order affects the order of the node creation and merging. The order of edge insertion is affected by the order in which the image graph is scanned and by the method used for the sorting. It is hard to extract the edges in some chaotic way, but using an unstable sorting method can easily lead to similar effect. Figure 1 shows the effect of stability of edge sorting on the tree construction from the BSD dataset. The x-axis shows the ratio of components of the α-tree to the number of components of the BPT, i.e. a rough estimate of the possible amount of redundant nodes. The y-axis shows the ratio of nodes removed during construction to the resulting number of nodes in the α-tree.

The edges were extracted row by row, first horizontal edges from a row, then vertical edges to the next row were extracted and so on. std::sort from libstdc++ (i.e., introsort) was used as the unstable sorting method. Figure 1 shows that the stability of sorting significantly affects the number of unnecessarily created nodes and unstable sorting methods should be avoided when possible. Fortunately, $O(N)$ methods such as counting-sort or

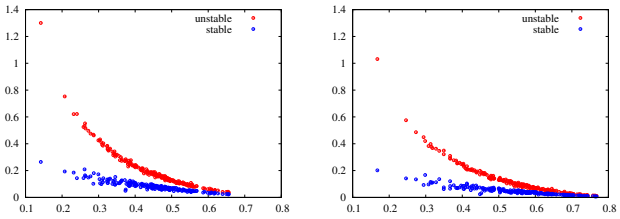


Fig. 1 Effects of stability of the edge sorting. X axis shows the ratio of α-tree node count to BPT node count, Y axis shows the ratio of unnecessary nodes to the final node count. l_∞ (left) and l_1 (right) norms are used for edge weights (BSD dataset).

radix-sort are stable and appropriate for sorting by edge weights. The regular order of edge insertion also leads to lower cache-trashing.

4.2 Tree construction timing

The performance of single-threaded tree construction was evaluated by constructing an α-tree from the 12 largest (by file size) images of the IHD dataset and the very large aerial image TOP using l_∞ norm (to keep the tree height at a reasonable level). We implemented and compared the following methods:

- BPT construction with a postprocessing step [23] (Najman);
- single-threaded construction from our previous work [12] (Havel);
- improved algorithm with *struct* tree (Struct);
- improved algorithm with *array* tree (Array);
- *array* tree with additional construction of child list for each node (Array+Chld).

The algorithm by Najman et al. was implemented on the same tree representation as the *Array* version of the new one. The code for edge extraction is shared and both algorithms also use the same improved union-find procedure. Therefore the measured performance should be comparable. The main difference is that our proposed the new algorithm tries to avoid construction of redundant nodes. The difference between *Struct* and *Havel* is that *Struct* uses root finding structure that is completely separate from the constructed tree.

The edge extraction and sorting steps use the same code for all methods. Their computation time is included in the total time for all algorithms but it should not affect the difference. Table 1 summarizes the time required to construct the α-trees. All times were obtained by averaging times of 10 runs. The time for the IHD dataset is average for all 12 images.

The results provide several interesting observations. Complicating the construction algorithm so it creates smaller amount of redundant nodes pays off (the difference between Najman and Array). The speedup seems to depend on the size of the input graph. More significant difference is between the representation by structs and arrays. The results clearly show that the added flexibility of the representation does come with a significant additional cost.

The last algorithm is significantly slower. This deserves deeper examination. The main difference is that the new algorithm uses the root finding structure completely separated from the constructed tree. Our previous work accesses the memory very sparsely and especially for large graphs, almost every memory access leads to a cache miss. Moving the root-finding structure outside of the tree seems to be a necessity.

Since the *Structs* and *Arrays* tree have significantly different capabilities, we included another test that constructs a full child list for the *arrays* so the tree could be traversed similarly to the *structs*. It was done by counting the children, using a prefix sum to calculate indices into the array of children and filling that array. It is interesting to notice that, even with these three additional passes, the tree construction is still faster than binary tree construction and postprocessing (this combination is called Array+Chld in Tab. 1).

4.3 Parallel tree construction

We finally evaluate the parallel tree (using *structs* data structure) construction on an SMT machine. 12 largest images of the IHD dataset were used to build an α-tree with l_∞ norm. The tests ran on an i72670QM processor with 8GB ram. The CPU has 4 cores with hyperthreading. Figure 2 shows relative speedups for increasing number of threads.

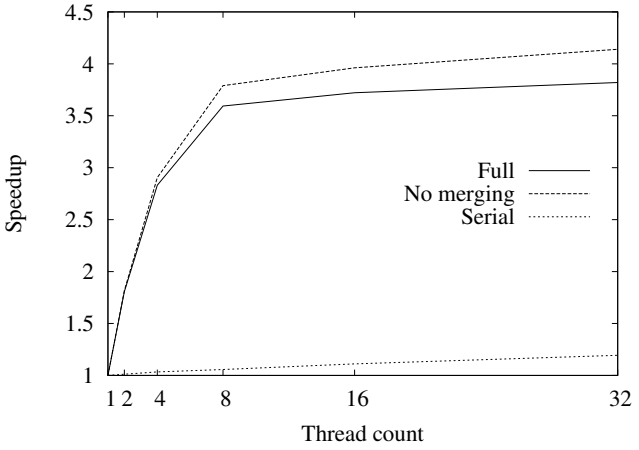


Fig. 2 Tree construction speedup for increasing thread count. Graph shows speedup for building partial trees alone, full tree construction and serialized version.

Three graphs are given in Fig. 2, to measure speedups respectively for the full tree construction and the con-

Algorithm	IHD time [s]	TOP time [s]
Array *	1.01	77.63
Array+Chld *	1.07	90.12
Najman [23]	1.10	93.24
Struct *	1.49	162.09
Havel [12]	2.49	665.57

Table 1 Performance of serial tree construction for IHD and TOP data. Contributions are marked with a star.

struction of partial trees only (without merging into the final tree). This roughly illustrates the maximal possible speedup that is limited by factors such as memory bandwidth. Third graph shows the speedup when the partial trees are constructed and merged in one thread. All graphs were generated by averaging measured speedups for individual images. Construction times were measured as an average of 10 attempts.

The parallel construction scales quite well with increasing number of threads. The flattening of speedup for more than 8 threads is not surprising due to number of virtual cores. However it is interesting that 8 threads perform significantly faster than 4. Since the virtual cores in hyperthreading share caches, for many parallel applications hyperthreading brings almost no performance gain. Here hyperthreading brings measurable improvement. Since the algorithm accesses memory chaotically, the lower effective amount of cache available for each virtual core does not affect the performance too much.

Third graph shows that construction and merging of partial trees has small positive effect even when one thread is used. This is probably caused by less cache failures when only parts of the image are processed at once.

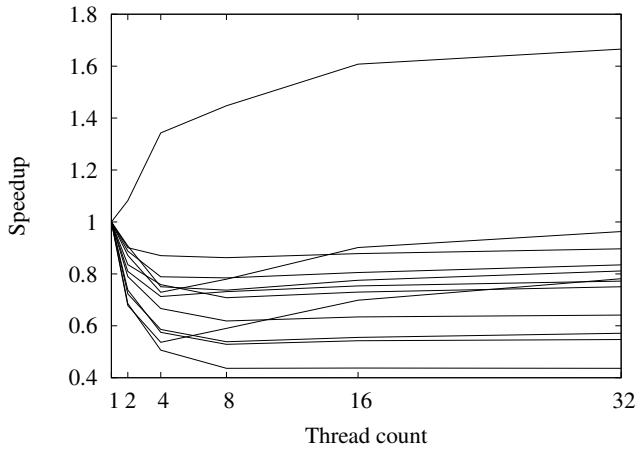


Fig. 3 Tree construction speedup when node size is not compared during merging. Each line shows speedup for one IHD image. Wrong merging of nodes can completely negate the results in degenerate cases.

As mentioned in previous section, it is critical to merge nodes so children of a smaller one are moved to larger one. Otherwise, for some degenerate images, the parallel construction can lead to a surprising slowdown. Figure 3 shows speedups for each image of the IHD dataset as a function of the thread count and illustrates this effect. The slowdown is caused by the fact that merging nodes requires $O(N)$ reconnection of child nodes of the absorbed node. Which node of a merged pair has more children depends on the shape of the merged trees.

4.4 Effect of tree height on parallelization

Since the tree merging in parallel construction requires bottom-up tree traversal without any possible path compression, the tree height significantly affects the merging complexity. However since the number of edges processed during merging is relatively low, the resulting performance impact is not obvious.

To get a very high tree, we extended the IHD images from 8b per channel up to 16b per channel with random low bits (similarly to [7]). In these synthetized data, the average tree height increased from roughly 80 for 8b images to 11000 for 16b images.

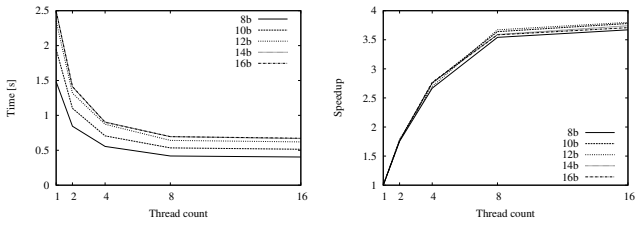


Fig. 4 Left : Tree construction times for different bit depths. Right : The relative speedups for different bit depths.

Figure 4 (left) shows the average times as a function of the number of threads for various bit depths. Figure 4 (right) shows the relative speedup with increasing thread count for different bit depths. While the tree construction slows down for increased image depth, parallel construction scales roughly the same for all image depths.

The tree depth does not affect the theoretical complexity of the singlethreaded construction, but it deffinitely affects the practical performance. Probably the main effect is a higher number of tree nodes which results of higher memory consumption and more cache misses. The tree merging complexity depends on the tree height, but it does not affect the scaling in practice. The low number

of edges that connect partial trees is one cause. Also the merging does not always traverse the whole tree. During lookup and zipping, the paths are not traversed independently and if they converge, the merging immediately ends. The gain from this optimization depents significantly on the tree structure, but we didn't find any degenerate cases where it impacts the performance in a negative way.

5 Applications

Beyond experiments held to evaluate the performance of the algorithms proposed in this paper (see Sec. 4), we address here the impact of efficient tree construction methods for modeling large visual data. To do so, we consider two different use cases, related to remotely-sensed image filtering and interactive video segmentation. These use cases are provided for illustration purpose only, thus explaining the lack of exhaustive comparison with the state-of-the-art.

5.1 VHR Remote Sensing

Earth observation through remote sensing technology benefits to a growing range of applications while facing new issues. Indeed, with the advent of very high resolution (VHR) sensors, satellite images are made of millions or even billions of pixels (called gigapixel images). The worldwide volume of such geospatial data is slowly reaching the Zettabyte scale. In this context, efficient means to access image content are required. Hierarchical structures like trees thus appear as a relevant model to represent such data.

Unsurprisingly, trees have been used in remote sensing to provide a multiscale description of image content (see Sec. 2.2 and [29,26,18,15,45,1]). Possible applications include land cover/land use mapping (through supervised classification), crisis management with damage assessment (change detection/classification) or image understanding (e.g., with object detection). Trees have been built upon panchromatic (from optical or SAR sensors) as well as multi- or hyperspectral images.

To illustrate the relevance of hierarchical representations in remote sensing, we consider the problem of image simplification, as already done by Brunner and Soille [6, 39] with α- and (α,ω)-connected components and trees. Image simplification aims at reducing the number of image elements to be processed, from elementary pixels to small regions or super pixels, and even to larger (and hopefully more meaningful) regions. As such, it of course eases image visualization, but it can also benefit to image compression or object-based image analysis (a paradigm that aims to consider regions as elementary units for image understanding in remote sensing). The interest of tree structures in this context relies in the fact that the scale (impacting the number or size of regions) is easily tunable. Indeed, changing the scale parameter used in image simplification only requires a new scan and cut of the tree, while other techniques need to perform a full analysis on the pixels of the image.

We rely here on an α-tree where each node is also embedded with its global range (to be compared with the ω threshold). Image simplification is achieved through a cut of the tree, given an ω threshold (set relatively to the highest global range observed in the image, i.e. its global dynamics). Furthermore, a size criterion is applied to discard nodes smaller than a given threshold. This additional constraint is easily verified for each leaf of the tree cut at the ω level. It results in pixels or nodes filtered out of the image. A simple post processing can be employed here to assign these elements to their closest neighboring node (based on edge values).

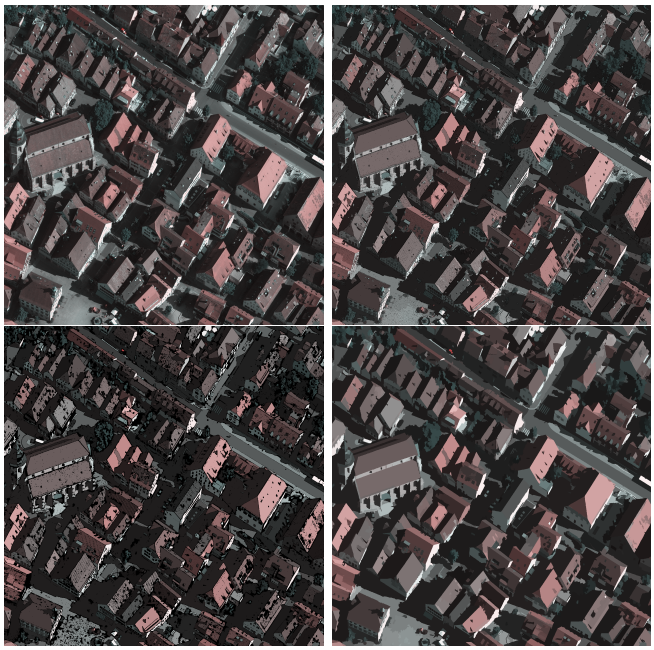
We apply this strategy on the Vaihingen dataset that has been provided through the ISPRS benchmark on urban object detection and 3D building reconstruction [10]. We consider here an aerial color images (16 bits) with a ground resolution below 10cm per pixel. The original data are provided together with a full mosaic of $20\,250 \times 21\,300$ pixels (called the TOP image in the previous section). For the sake of clarity, we show here only a part ($2\,000 \times 2\,000$ pixels) of the different images instead of the full original ones.

Simplification of the aerial color images is given in Fig. 5, where all images have been converted to 8-bit for visualization. We can visually observe the effect of the simplification process, that can also be measured quantitatively in terms of image elements: from 4 000 000 pixels in the original image, the simplification process returns 108 135 nodes (low area threshold, $\mathcal{A} > 1$) or even only 2 583 nodes (high area threshold, $\mathcal{A} > 100$). In other words, a compression ratio of more than 1:1 500 is achieved without no significant loss in terms of visible objects of interest. This allows subsequent steps to be much more scalable as well as relying on more meaningful information.

Finally, let us recall that, contrary to [6,39], the computation time required by the simplification process is here not increasing with the scale but it is scale-independent. This is due to the method involved, i.e. a single cut of the tree instead of an iterative process, the tree having been computed offline very efficiently (e.g., see Tab. 1) with the algorithms presented in Sec. 3.

5.2 Interactive Object Selection in Video Sequences

Another kind of visual data characterized by a large or very large volume is video sequences. Indeed, a video file is made of many frames or images acquired at a given framerate (most often 25 frames per second or fps). A simple 1 hour-long video sequence with VGA resolution (e.g., 640×480 pixels) and standard framerate (25 Hz) contains more than 27 billions of pixels.



(a) (b) (c) (d)

Fig. 5 Simplification of an aerial color image (a) with ω threshold set to a tenth of the maximal observed global range, and various area thresholds: $\mathcal{A} > 1$ (b) and $\mathcal{A} > 100$ (c). Nodes filtered out are shown in black. Following a significant filtering $\mathcal{A} > 100$, they are assigned to their closest neighbor (d). This removes small details and "flattens" larger areas. Please zoom the pdf to see the details.

Such data require efficient management tools to be handled, e.g. through interactive multimedia editing techniques. A common process is to extract objects from videos, edit them and finally reintroduce the modified objects within the original video or another one. To allows this kind of video editing, interactive object selection techniques are needed. Many approaches have been proposed in the last decade [5,47,4,2,33,11,25]. Nevertheless, as shown by results achieved by the most recent techniques (e.g., [48] allows a processing framerate of 1.3-1.5 fps, far below current video broadcast standards), there is still a lot of effort to be achieved in terms of computational efficiency. Even existing parallel strategies (e.g., [11]) do not allow for processing realistic videos (20 minutes are required to process a 40s video).

In this context, trees appear as a relevant solution and first results were promising [19]. We propose here to built upon our previous work [19] and we define the following strategy for interactive object selection.

First, a single tree is built from a video sequence, assuming space-time connectivity and representing the video as a spatio-temporal volume. Each pixel is then defined by a triplet (x, y, t) with two spatial and one temporal coordinates. We define here the neighborhood using the 6-connectivity i.e., two pixels (x, y, t) and (x', y', t') are neighbors if $|x - x'| + |y - y'| + |t - t'| = 1$. The similarity measure is simply the Euclidean norm computed

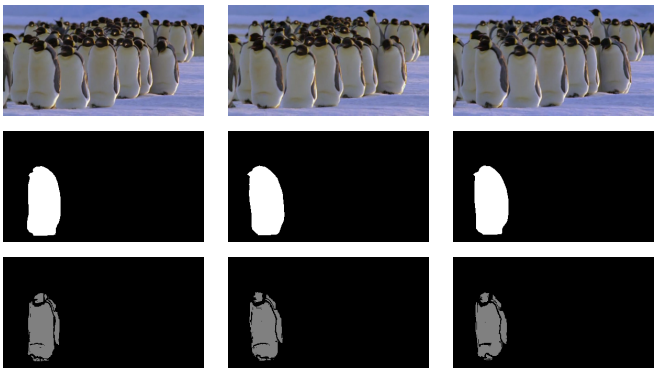


Fig. 6 Illustrative results of the tree-based object selection technique with the penguin video from SegTrack. From top to bottom: sample frames from the original sequence, associated ground truth, and extracted objects.

in the RGB color space. Of course other metrics could have been used, but let us recall that our goal here is not to provide the most accurate results, but rather to illustrate the potential of tree representations for analyzing and processing large datasets. Furthermore, preliminary experiments achieved with the Lab color space did not lead to better results. Having defined the neighborhood and local similarity measure, the construction of the tree is straightforward using algorithms discussed in Sec. 3.

Following its computation, the tree becomes the image representation to be further processed (instead of the initial array of pixels). The interactive object selection scheme proposed here relies on the definition by the user of an object of interest. To do so, a user specifies a rectangular selection through mouse clicking on any frame of the video sequence (for better accuracy, it is possible to set the selection in several frames simultaneously). We assume that the object of interest is completely included in the selection made by the user. This object (i.e., foreground) is then distinguished from the other parts of the video (i.e., background). To do so, we remove all background nodes from the set of nodes included in the user selection, under the assumption that background nodes span outside the selection. Furthermore, noise robustness is ensured by filtering the small nodes, i.e. nodes with temporal duration below a given threshold.

To evaluate the relevance of such an approach, we consider a standard dataset and compare our results with the state-of-the-art. More precisely, we use the SegTrack dataset [44] that contains 6 video sequences, with length of 20–70 images of rather small size (from 320×240 pixels to 414×352 pixels), together with a reference segmentation (ground truth).

For the sake of illustration, we compare our method with one of the most recent works relying on this dataset [48] that requires a light input from the user, similarly to our method. Comparative results are provided in Tab. 2 using the same evaluation metrics than in [48]. Unsurprisingly, the basic selection scheme described in this section (without any learning/modeling of the object,

nor any usage of motion/texture information) produces a higher error score than [48]. But on some challenging videos like *penguin*, preliminary results are satisfactory as shown by Fig. 6, while many state-of-the-art techniques fail in this situation (with many similar objects) [14,32]. Indeed, more complex schemes for tree analysis still have to be explored and are out-of-the-scope of this paper.

A careful analysis of the weak results obtained by the proposed method with the Girl sequence leads to the following observation. This video sequence is characterized by some important motion of the objects, whose visual impact consists in the lack of temporal connectivity between the moving objects in the successive frames of the sequence. Besides, the video content is also blurred due to this motion effect. Furthermore, this video is characterized by a strong visual artefact consisting in an superimposed watermark. With the α-tree model considered in this experiment, this leads to some spatio-temporal chaining effects that cannot be overcame without any post-processing or constrained imposed on the connectivity between pixels. Let us note that the artefacts due to high motion can be alleviated with high-speed videos. In this case, the proposed strategy looks very appealing due to its low complexity.

In spite of a loss in quality, the proposed tree model appears as an efficient image/video representation. As shown in Tab. 2, it offers a great improvement in terms of efficiency (ca. 3 orders of magnitude) over [48], which is one of the most efficient semi-supervised segmentation techniques (while unsupervised techniques are even more computationally expensive). It thus allows for introduction of more complex (but still tree-based) image/video analysis strategies, to ensure both high quality and efficiency. To illustrate, let us recall that many video segmentation methods rely on a first segmentation into superpixels, that might be easily produced by cutting the tree. But while extracting superpixels from a SegTrack video requires at least 500 seconds with the most efficient techniques from the state-of-the-art [51] (measured with a Dual Quad-core Intel Xeon CPU E5620 2.4 GHz, 16GB RAM running Linux), our algorithm is able to provide a set of superpixels in less than 50 seconds with a Java implementation and a standard laptop configuration (Dual-core Intel Core CPU i5-3320M 2.60GHz, 6GB RAM running Linux).

6 Conclusion

In this paper, we consider multiscale image representation and analysis through tree models originated from Mathematical Morphology. Contrarily to component tree that have been extensively studied [8], but requires to set an ordering relation on the data (i.e. pixel values), we focus here on the α tree that relies on the edge values instead of pixel (vertex) ones. More precisely, two recent α-tree construction algorithms were combined and incrementally inproved. This improved algorithm combines efficient single-threaded construction with the possibility of parallelization and outperforms both. For the sake of research reproducibility and to support the use of the α-tree model and algorithms in various applications of image processing and computer vision, we have made the code available on Github[1].

We also reviewed two possible data structures for tree representations, and compared them in terms of performance and flexibility. The representation by arrays of integers is significantly simpler and more efficient, but is not as flexible as dynamic structures linked by pointers. The difference is so significant, arrays should be used whenever possible, but the structures allow greater flexibility.

Finally, we have illustrated the possible interest of such tree models on several use cases, dealing with remote sensing image analysis and video segmentation. We have shown that the tree structure provides a very efficient way to process and analyse images of various natures.

While we show here how parallel construction of α-tree can be achieved with dynamic structures, designing such a solution with arrays is still very challenging, contrarily to the case of min/max-tree. Indeed, efficient merging of such trees remains an open problem.

The merging of partial trees can be applied in many possible ways. The applications in this paper cover only a small subset. We plan to try this approach on other problems with large data (e.g., 3D volumes, long video sequences). In particular, such a strategy can be used for online processing of video streams to overcome the limitations related to memory requirements.

Acknowledgements

The authors would like to thank Michael Wilkinson for fruitful discussions and valuable suggestions, which helped in improving the manuscript.

The Vaihingen data set was provided by the German Society for Photogrammetry, Remote Sensing and Geoinformation (DGPF) [10].

References

1. A. Alonso-Gonzalez, S. Valero, J. Chanussot, C. Lopez-Martinez, and P. Salembier. Processing multidimensional sar and hyperspectral images with binary partition tree. *Proceedings of the IEEE*, 101(3):723–747, 2013.
2. X. Bai, J. Wang, D. Simons, and G. Sapiro. Video snapcut: robust video object cutout using localized classifiers. In *Proceedings of the SIGGRAPH*, 2009.

[1] https://github.com/jirihavel/libcct

Video sequence	Proposed technique		Wang et al. [48]	
	Error	Efficiency	Error	Efficiency
Birdfall	0.006 (497)	0.025	0.003 (248)	20.000
Parachute	0.014 (2,022)	0.042	0.002 (228)	35.000
Girl	0.079 (10,067)	0.142	0.013 (1,691)	16.000

Table 2 Comparative results for interactive object selection. Error is expressed as the average fraction (and absolute number in parentheses) of mis-segmented pixels (false positive plus false negative) per frame. Efficiency is computed in seconds considering an execution on a standard laptop with a Java application, and recalled from [48].

3. P. Bosilj, S. Lefèvre, and E. Kijak. Hierarchical image representation simplification driven by region complexity. In *International Conference on Image Analysis and Processing*, pages 562–571, 2013.
4. Y. Boykov and G. Funka-Lea. Graph cuts and efficient n-d image segmentation. *International Journal of Computer Vision*, 70(2):109–131, 2006.
5. Y. Boykov and M.P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. In *Proceedings of the ICCV*, pages 105–112, 2001.
6. D. Brunner and P. Soille. Iterative area filtering of multichannel images. *Image and Vision Computing*, 25(8):1352–1364, August 2007.
7. E. Carlinet and T. Géraud. A comparison of many max-tree computation algorithms. In *In Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 73–85. Springer, 2013.
8. E. Carlinet and T. Géraud. A comparative review of component tree computation algorithms. *IEEE Transactions on Image Processing*, 23(9):3885–3895, September 2014.
9. J. Cousty, L. Najman, and B. Perret. Constructive links between some morphological hierarchies on edge-weighted graphs. In *International Symposium on Mathematical Morphology*, pages 135–146, 2013.
10. M. Cramer. The dgpf test on digital aerial camera evaluation – overview and test design. *Photogrammetrie – Fernerkundung – Geoinformation*, 2:73–82, 2010.
11. M. Grundmann, V. Kwatra, M. Han, and I. Essa. Efficient hierarchical graph based video segmentation. *IEEE CVPR*, 2010.
12. J. Havel, F. Merciol, and S. Lefèvre. Efficient schemes for computing α-tree representations. In *International Syposium on Mathematical Morphology*, pages 111–122, 2013.
13. H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometry consistency for large scale image search. In *Proceedings of the 10th European conference on Computer vision*, 2008.
14. Y.J. Lee, J. Kim, and K. Grauman. Key-segments for video object segmentation. In *ICCV*, 2011.
15. S. Lefèvre, L. Chapel, and F. Merciol. Hyperspectral image classification from multiscale description with constrained connectivity and metric learning. In *6th International Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2014.
16. D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings of the 8th International Conference on Computer Vision*, volume 2, pages 416–425, Vancouver, Canada, July 2001.
17. P. Matas, E. Dokladalova, M. Akil, V. Georgiev, and M. Poupa. Parallel hardware implementation of connected component tree computation. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 64 –69, 31 2010-sept. 2 2010.
18. F. Merciol, L. Chapel, and S. Lefèvre. Hyperspectral image representation through α-trees. In *ESA-EUSC-JRC Conference on Image Information Mining*, pages 37–40, 2014.
19. F. Merciol and S. Lefèvre. Fast image and video segmentation based on α-tree multiscale representation. In *International Conference on Signal Image Technology and Internet Based Systems*, pages 336–342, Naples, Italy, November 2012.
20. P. Monasse. Contrast invariant registration of images. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 6, pages 3221–3224, 1999.
21. P. Monasse and F. Guichard. Fast computation of a contrast-invariant image representation. *IEEE Transactions on Image Processing*, 9(5):860–872, 2000.
22. L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11):3531–3539, 2006.
23. L. Najman, J. Cousty, and B. Perret. Playing with kruskal: Algorithms for morphological trees in edge-weightted graphs. In *International Syposium on Mathematical Morphology*, pages 135–146, 2013.
24. D. Nister and H. Stewenius. Linear time maximally stable extremal regions. In *ECCV*, pages 183–196, 2008.
25. A. Noma, A. Graciano, R. Cesar Jr, L. Consularo, and I.Bloch. Interactive image segmentation by matching attributed relational graphs. *Pattern Recognition*, 45:1159–1179, 2012.
26. G. Ouzounis and L. Gueguen. Interactive collection of training samples from the max-tree structure. In *IEEE International Conference on Image Processing*, pages 1449–1452, 2011.
27. G. Ouzounis, V. Syrris, L. Gueguen, and P. Soille. The switchboard platform for interactive image information mining. In P. Soille, M. Iapaolo, P.G. Marchetti, and M. Datcu, editors, *Proc. of 8th Conference on Image Information Mining*, pages 26–30. ESA-EUSC-JRC, October 2012.
28. G. Ouzounis and M. Wilkinson. Mask-based second-generation connectivity and attribute filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):990–1004, 2007.
29. G. K. Ouzounis and P. Soille. Pattern spectra from partition pyramids and hierarchies. In *International Symposium on Mathematical Morphology*, pages 108–119, Verbania-Intra, Italy, 2011.
30. N. Passat and B. Naegel. Selection of relevant nodes from component-trees in linear time. In *IAPR International Conference on Discrete Geometry for Computer Imagery*, pages 453–464, 2011.
31. N. Passat, N. Naegel, F. Rousseau, M. Koob, and J.L. Dietemann. Interactive segmentation based on component-trees. *Pattern Recognition*, 44(10–11):2539–2554, 2011.
32. Sebastien Poullot and Shin'Ichi Satoh. Vabcut: a video extension of grabcut for unsupervised video foreground object segmentation. In *VISAPP*, 2014.

33. B. Price, B. Morse, and S. Cohen. Livecut: Learning-based interactive video segmentation by evaluation of multiple propagated cues. In *IEEE International Conference on Computer Vision*, 2009.
34. P. Salembier and L. Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *IEEE Transactions on Image Processing*, 9(4):561–576, 2000.
35. P. Salembier, A. Oliveras, and L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.
36. J. Serra. Anamorphoses and function lattices. In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing"*, chapter 13, pages 483–523. Marcel Dekker, New York, 1993.
37. J. Serra. The "false colour" problem. In *International Symposium on Mathematical Morphology*, pages 13–23, Groningen, The Netherlands, August 2009.
38. J. Serra and B. Kiran. Optima on hierarchies of partitions. In *International Symposium on Mathematical Morphology*, pages 147–158, 2013.
39. P. Soille. Constrained connectivity for hierarchical image partitioning and simplification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(7):1132–1145, July 2008.
40. P. Soille and J. Grazzini. Constrained connectivity and transition regions. In *International Symposium on Mathematical Morphology*, pages 59–69, Groningen, The Netherlands, August 2009.
41. Y. Song and A. Zhang. Analyzing scenery images by monotonic tree. *ACM Multimedia Systems*, 8(6):495–511, 2003.
42. S. Crozet T. Géraud, E. Carlinet and L. Najman. A quasi-linear algorithm to compute the tree of shapes of nd images. In *Mathematical Mor- phology and Its Applications to Signal and Image Processing*, pages 98–110. Springer, 2013.
43. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
44. D. Tsai, M. Flagg, and J.M. Rehg. Motion coherent tracking with multi-label mrf optimization. *British Machine Vision Conference*, 2010.
45. S. Valero, P. Salembier, and J. Chanussot. Hyperspectral image representation and processing with binary partition trees. *IEEE Transactions on Image Processing*, 22(4):1430–1443, 2013.
46. V. Vilaplana, F. Marques, and P. Salembier. Binary partition trees for object detection. *IEEE Transactions on Image Processing*, 17(11):2201–2216, 2008.
47. J. Wang, P. Bhat, R.A. Colburn, M. Agrawala, and M.F. Cohen. Interactive video cutout. *ACM Transactions on Graphics*, 24(3):585–594, 2005.
48. T. Wang, B. Han, and J. Collomosse. Touchcut: Fast image and video segmentation using single-touch interaction. *Computer Vision and Image Understanding*, 120:14–30, 2014.
49. J. Weber, S. Lefèvre, and P. Gançarski. Interactive video segmentation based on quasi-flat zones. In *International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 265–270, Dubrovnik, Croatia, September 2011.
50. Michael H. F. Wilkinson, Hui Gao, Wim H. Hesselink, Jan-Eppo Jonker, and Arnold Meijster. Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(10):1800–1813, 2008.
51. C. Xu and J. J. Corso. Evaluation of super-voxel methods for early video processing. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2012.